

Refining Trace Abstraction using Abstract Interpretation

Marius Greitschus, Daniel Dietsch, and Andreas Podelski

University of Freiburg, Germany

Abstract. The CEGAR loop in software model checking notoriously diverges when the abstraction refinement procedure does not derive a loop invariant. An abstraction refinement procedure based on an SMT solver is applied to a trace, i.e., a restricted form of a program (without loops). In this paper, we present a new abstraction refinement procedure that aims at circumventing this restriction whenever possible. We apply abstract interpretation to a program that we derive from the given trace. If the program contains a loop, we are guaranteed to obtain a loop invariant. We call an SMT solver only in the case where the abstract interpretation returns an indefinite answer. That is, the idea is to use abstract interpretation and an SMT solver in tandem. An experimental evaluation in the setting of trace abstraction indicates the practical potential of this idea.

1 Introduction

When trying to prove the correctness of a program, finding useful abstractions in form of state assertions is the most important part of the process [15, 21]. In this context, usefulness is about being able to prove correctness as efficiently as possible. Hence, in order to be able to analyze large programs, it is important to find state assertions automatically. The conflict between these two goals gives rise to different techniques for synthesizing state assertions. For example, abstract interpretation [11] is a well-known method for finding state assertions. Abstract interpretation computes an over-approximation of a program's states by using an up-front and largely program-independent abstraction. Many such abstractions exist ([12, 23, 24, 7]) and all of them are useful, because they give rise to different kinds of state assertions that can be used to prove the correctness of different kinds of programs. It is the strength of abstract interpretation that it always terminates and always computes a fixpoint in the selected abstraction. If the program contains loops, the fixpoint computed for the loop head is also, by definition, a loop invariant which allows for an easy abstraction of loops. While abstract interpretation scales favorably with the size of the program, the computed over-approximation is often not precise enough to be useful to prove the correctness of a program.

Another example to address this task is to use software model checking tools like BLAST [6], SLAM [3], and more recently, CPACHECKER [9] and ULTIMATE AUTOMIZER [17], that follow the counterexample-guided abstraction refinement

(CEGAR) approach [10]. In CEGAR, an abstraction is continuously refined by synthesizing state assertions from paths through the control flow graph of the program that 1) are not contained in the current abstraction, 2) can reach an error location, and 3) are not executable. By extracting state assertions from those paths, the abstraction can be refined to fit the program at hand, which allows the user a greater amount of flexibility in choosing her programs. Because the path analysis has to be precise, i.e., it has to ensure that paths that represent real errors can be identified, it often produces state assertions that are too strong to be loop invariants, in turn forcing the CEGAR algorithm to unroll loops of the program. If this happens, the algorithm may not be able to refine the abstraction at all, e.g., because the loop of the analyzed program can be unrolled infinitely often.

In this paper we propose a unification of both techniques, abstract interpretation and CEGAR-based software model checking, such that both can benefit from their strengths: we use abstract interpretation to find loop invariants, an interpolating SMT solver to analyze single paths, and we combine both in a CEGAR-based abstraction refinement loop.

1.1 Example

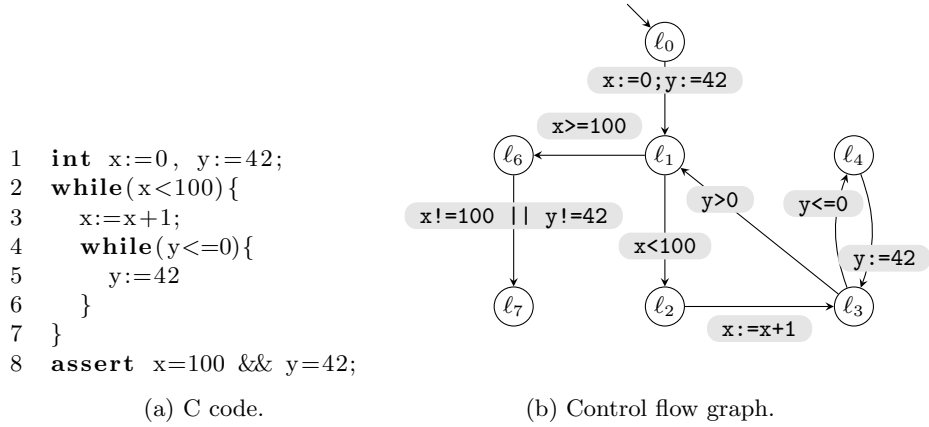


Fig. 1: Example program \mathcal{P}_1 with its C code and its corresponding control flow graph (CFG). The location ℓ_0 of the CFG is the initial location, ℓ_7 is the error location.

Consider the example program \mathcal{P}_1 in Figure 1 and its corresponding control flow graph. We are interested in proving that the error location of \mathcal{P}_1 's control flow graph (see Figure 1b) ℓ_7 is unreachable. A CEGAR-based approach to generate the proof by iteratively refining an abstraction of the program begins with picking a sequence of statements from the CFG, which starts in the initial

location and ends in an error location. Next, an analysis decides whether the selected sequence of statements is executable or not, and if not, the abstraction is refined such that this particular sequence is no longer contained.

Consider the shortest sequence of statements τ_1 from the initial location ℓ_0 to the error location ℓ_7 .

$\tau_1 : \text{ x:=0;y:=42 } \text{ x}>=100 \text{ } \text{ x!=100 || y != 42 }$

This sequence of statements is not executable, because the first two statements are contradicting each other. A possible proof for this contradiction consists of the following sequence of assertions.

true x:=0;y:=42 **x=0** $\text{ x}>=100$ **false** $\text{ x!=100 || y != 42 }$ **false**

This sequence of state assertions allows the CEGAR tool to refine its abstraction such that τ_1 is removed. In the next iteration, we assume that τ_2 is selected.

$\tau_2 : \text{ x:=0;y:=42 } \text{ x}<100 \text{ } \text{ x:=x+1 } \text{ y}>0 \text{ } \text{ x}>=100 \text{ } \text{ x!=100 || y != 42 }$

Again, this sequence of statements is not executable. For example, the statements x:=0;y:=42 , x:=x+1 and $\text{ x}>=100$ contradict each other, for which we can extract the following proof.

true x:=0;y:=42 **x=0** $\text{ x}<100$ **x=0** x:=x+1 **x=1** $\text{ y}>0$
x=1 $\text{ x}>=100$ **false** $\text{ x!=100 || y != 42 }$ **false**

We can continue in this fashion until we have unrolled the outer while loop of \mathcal{P}_1 , but we would rather find other proofs that contain state assertions that allow us to find a more general refinement of our abstraction, thus eliminating the need for unrolling.

In our example, the state assertions are not general enough to efficiently prove the program's correctness, although they were obtained using a state-of-the-art interpolating SMT solver. The reason we obtain such assertions is that most solvers prefer to find proofs for contradictions with as few clauses as possible. A more useful but larger reason would involve using the statement $\text{ x!=100 || y != 42 }$, where the SMT solver needs to construct a proof of unsatisfiability that contains both clauses, instead of using $\text{ x } >= 100$, where only one clause has to be contradicted. With the help of that larger statement the state assertion $\text{ x } <= 100$ could be obtained. $\text{ x } <= 100$ is very useful because it is a loop invariant at location ℓ_1 , and together with the easily obtained invariant $\text{ y } = 42$, the two state assertions are sufficient to prove the correctness of our example.

Approaches based on static program analysis, such as abstract interpretation, can deduce loop invariants by computing a fixpoint for each program location. However, such an analysis of the whole program may not be able to find an invariant strong enough to prove the program to be correct.

One way of improving the precision is not analyzing the whole program but just a fragment of it. We can compute such a fragment by projecting the CFG

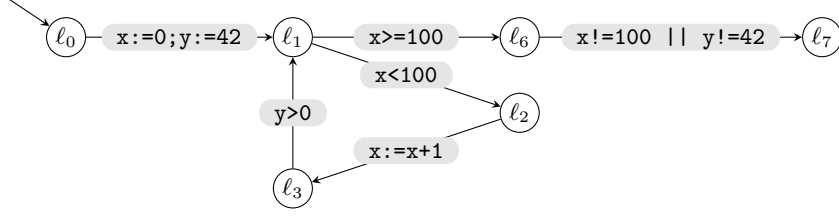


Fig. 2: The path program computed from the sequence of statements τ_2 .

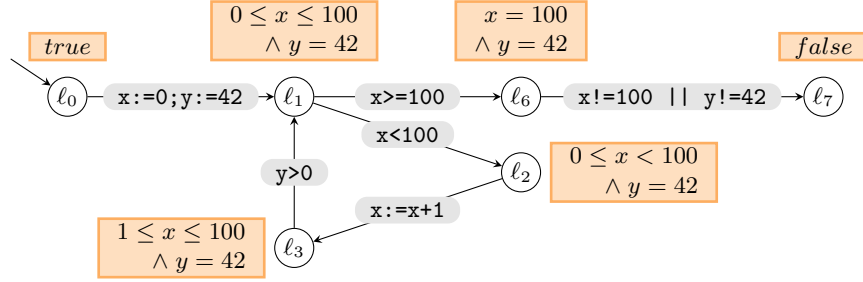


Fig. 3: State assertions computed by an interval analysis on the path program from Figure 2.

of the program to the statements from the selected sequence of statements. The resulting CFG is called a *path program* [8]. Figure 2 shows the path program computed from \mathcal{P}_1 and the sequence of statements τ_2 . We can now calculate the fixpoint of, e.g., an interval abstraction for this path program, which then yields the state assertions shown in Figure 3. In this case, the state assertion at ℓ_1 already contains the desired loop invariant in the second CEGAR iteration. In general, an interval abstraction may not be enough to find a useful invariant, but refining the abstraction with conventional methods still allows the overall algorithm to progress.

In the following, we present our approach that combines the analysis of single sequences of statements and the analysis of path programs in an automata-theoretic setting. We focus on obtaining loop invariants with abstract interpretation if possible, but can fall back on the analysis of single traces if the computed abstraction is too weak to prove infeasibility of a trace.

2 Preliminaries

In this section, we present our understanding of programs and their semantics, give a brief overview over abstract interpretation, and explain the trace abstraction algorithm which we use as basis of our approach.

Programs and Traces. We consider a simple programming language whose statements are assignment, assume, and sequential composition. We use the syntax that is defined by the following grammar

$$s := \text{assume } bexpr \mid x := expr \mid s; s$$

where Var is a finite set of program variables, $x \in Var$, $expr$ is an expression over Var and $bexpr$ is a Boolean expression over Var . For brevity we use $bexpr$ to denote the assume statement $\text{assume } bexpr$.

We represent a *program* over a given set of statements $Stmt$ as a labeled graph $\mathcal{P} = (Loc, \delta, \ell_0)$ with a finite set of nodes Loc called locations, a set of edges labeled with statements, i.e., $\delta \subseteq Loc \times Stmt \times Loc$, and a distinguished node ℓ_0 which we call the initial location.

We call a sequence of statements $\tau = s_0 s_1 s_2 \dots \in Stmt^*$ a *trace of the program* \mathcal{P} if τ is the edge labeling of a path that starts at the initial location ℓ_0 . We define the set of all program traces formally as follows:

$$T(\mathcal{P}) = \{s_0 s_1 \dots \in Stmt^* \mid \exists \ell_1, \ell_2, \dots \bullet (\ell_i, s_i, \ell_{i+1}) \in \delta, \text{ for } i \geq 0\}$$

Note that in each program trace, the source location of the edge labeled with s_0 is the initial location, and therefore, ℓ_0 is not existentially quantified in the formula for $T(\mathcal{P})$.

Let \mathcal{D} be the set of values of the program's variables. We denote a program state σ as a function $\sigma : Var \rightarrow \mathcal{D}$ that maps program variables to values. We use S to denote the set of all program states. Each statement $s \in Stmt$ defines a binary relation ρ_s over program states which we call the *successor relation*. Let $Expr$ be the set of all expressions over the program variables Var . We assume a given interpretation function $\mathcal{I} : Expr \times (Var \rightarrow \mathcal{D}) \rightarrow \mathcal{D}$ and define the relation $\rho_s \subseteq S \times S$ inductively as follows:

$$\rho_s = \begin{cases} \{(\sigma, \sigma') \mid \mathcal{I}(bexpr)(\sigma) = true \text{ and } \sigma = \sigma'\} & \text{if } s \equiv \text{assume } bexpr \\ \{(\sigma, \sigma') \mid \sigma' = \sigma[x \mapsto \mathcal{I}(expr)(\sigma)]\} & \text{if } s \equiv x := expr \\ \{(\sigma, \sigma') \mid \exists \sigma'' \bullet (\sigma, \sigma'') \in \rho_{s_1} \text{ and } (\sigma'', \sigma') \in \rho_{s_2}\} & \text{if } s \equiv s_1; s_2 \end{cases}$$

Given a trace $\tau = s_0 s_1 s_2 \dots$, a sequence of program states $\pi = \sigma_0 \sigma_1 \sigma_2 \dots$ is called a *program execution of trace* τ if each successive pair of program states is contained in the successor relation of the corresponding statement of the trace, i.e., $(\sigma_i, \sigma_{i+1}) \in \rho_{s_i}$ for $i \in \{0, 1, \dots\}$. We call a trace τ *infeasible* if it does not have any program execution, otherwise we call τ *feasible*. We use $\Pi(\tau)$ to denote the set of all program executions of τ . The set of all feasible traces of program \mathcal{P} is denoted by $T_{feas}(\mathcal{P})$, and the set of all program executions of \mathcal{P} , $\Pi(\mathcal{P})$, is defined as follows.

$$\Pi(\mathcal{P}) = \bigcup_{\tau \in T_{feas}(\mathcal{P})} \Pi(\tau)$$

Abstract Interpretation Abstract interpretation [11] is a well-known static analysis technique that computes a fixpoint of abstract values of an input program’s variables for each program location. This fixpoint is an over-approximated abstraction of the program’s concrete behavior. To this end, abstract interpretation uses an *abstract domain* defining allowed abstract values of the program’s variables in the form of a complete lattice. The fixpoint computation algorithm analyzes an input program and annotates each location with an abstract state by iteratively applying an abstract transformer for each edge, starting at the initial location. This abstract transformer computes an abstract post state for a given abstract state and a statement, i.e., it computes the effect a statement has on a given abstract state. In case of branching in the program, the fixpoint computation algorithm may choose to either merge the states at the join point of the branches with a join operator defined by the abstract domain, or to keep an arbitrary number of disjunctive states. In the latter case, precision is increased at the cost of additional computations due to more abstract states in the abstraction.

The fixpoint computation algorithm is guaranteed to achieve progress and to eventually terminate, making abstract interpretation one of the most scalable approaches for program analysis. Upon termination, an over-approximated abstraction of the program is guaranteed to have been computed. Progress is achieved by the application of a widening operator, defined by the used abstract domain. When the fixpoint computation algorithm traverses the statements of a loop, an infinite repetition of the application of the abstract transformer to the loop’s statement is avoided by widening the approximation of the loop’s body. This way, the approximation of the loop is made increasingly wider until a fixpoint for the effect of the whole loop is found.

Trace Abstraction. The trace abstraction algorithm [19, 20] is a CEGAR-based software model checking approach that proves the correctness of a program \mathcal{P} by partitioning the set of possible error traces in feasible and infeasible traces. In the following, we briefly explain this approach. Consider the trace abstraction algorithm shown in Figure 4. The input program \mathcal{P} over the set of statements $Stmt$ is first translated into a *program automaton* $\mathcal{A}_{\mathcal{P}}$, which encodes the correctness property of \mathcal{P} by marking some of its locations as error locations. Those error locations serve as the accepting states of the program automaton $\mathcal{A}_{\mathcal{P}}$, and the set of statements $Stmt$ as its alphabet Σ . By construction, every word accepted by this automaton represents a trace of \mathcal{P} that can reach the error location. Next, the algorithm determines whether the language of $\mathcal{A}_{\mathcal{P}}$ contains a feasible trace, which would then be a valid counterexample. To this end, a *data automaton* \mathcal{A}_D over the same alphabet as $\mathcal{A}_{\mathcal{P}}$ is constructed such that its language consists only of infeasible traces. More formally, a data automaton is a Floyd-Hoare automaton [20, 14]. A Floyd-Hoare automaton $\mathcal{A} = (Q, \delta, q_0, F)$ is an automaton over the alphabet of the program’s statements $Stmt$ together with a mapping that assigns to each state $q \in Q$ a formula φ_q that denotes a predicate over the program variables such that the following holds:

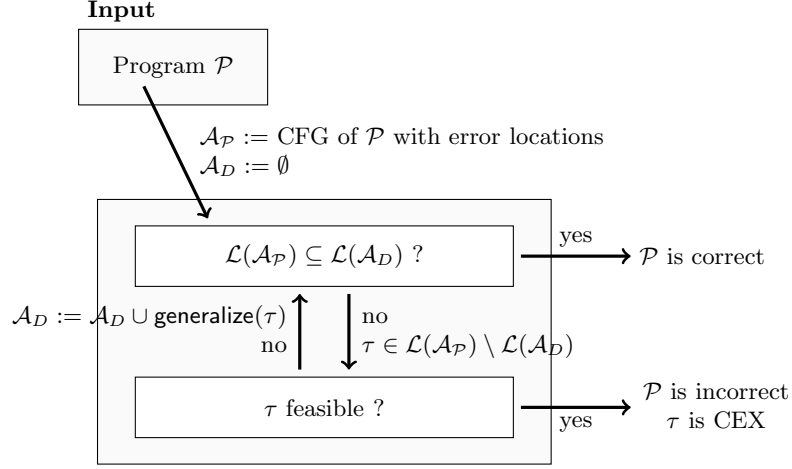


Fig. 4: The trace abstraction algorithm.

- The initial state is annotated by the formula *true*.
- For each transition $(q, st, q') \in \delta$ the triple $\{\varphi_q\} \ st \ \{\varphi_{q'}\}$ is a valid Hoare triple.
- Each accepting state $q \in F$ is annotated by the formula *false*.

Initially, the data automaton \mathcal{A}_D is empty. In each iteration, the algorithm checks whether the language of the current data automaton \mathcal{A}_D is a superset of the language of the program automaton \mathcal{A}_P . If this is the case, all traces in \mathcal{A}_P are infeasible, i.e., the error locations of program \mathcal{P} cannot be reached. If this is not the case, there exists a trace τ of \mathcal{A}_P which is not in \mathcal{A}_D , and thus not known to be infeasible.

Therefore, if the trace τ is feasible, it represents at least one valid program execution that can reach an error location. If the trace τ is infeasible, the algorithm constructs a new data automaton \mathcal{A}_D whose language contains more infeasible traces than the old by computing a union of the old automaton \mathcal{A}_D and a new automaton obtained by generalizing the proof of infeasibility of τ (*generalize*).

3 Algorithm

In this section we present a modified version of the trace abstraction CEGAR loop introduced in Section 2, which uses a new method based on abstract interpretation for obtaining the data automaton. Our algorithm is shown in Figure 5. As in the default trace abstraction algorithm, the input program \mathcal{P} is translated into a program automaton \mathcal{A}_P . The initial data automaton \mathcal{A}_D is empty.

After analyzing a possible counterexample trace τ for infeasibility, we first construct a *path program* $\mathcal{P}_\tau^\#$ from the trace. We use the theoretical foundations

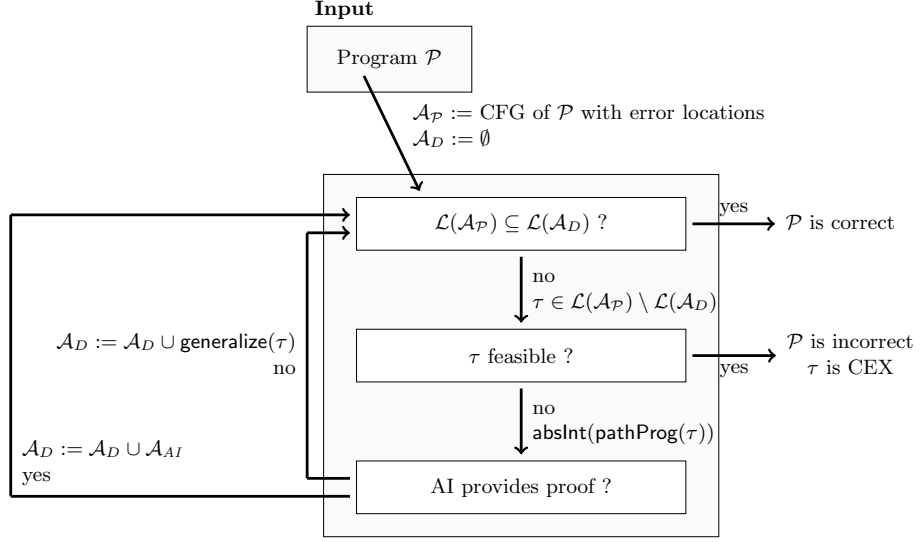


Fig. 5: The trace abstraction algorithm with abstract interpretation refinement.

of path programs provided by Beyer et al. [8]. In our context, a path program is defined as follows.

- A path program $\mathcal{P}^\#$ of program $\mathcal{P} = (Loc, \delta, \ell_0)$ with statements $Stmt$ and trace $\tau = s_0 s_1 \dots \in 2^{Stmt}$ of \mathcal{P} is a program $\mathcal{P}^\# = (Loc^\#, \delta^\#, \ell_0^\#)$ such that
- the set of program locations $Loc^\#$ contains only locations that are visited by trace τ , i.e., $Loc^\# = \{\ell \mid \ell \in Loc \wedge \exists s_i \in \tau \text{ s.t. } (\ell, s_i, \ell') \in \delta \vee (\ell', s_i, \ell) \in \delta\}$,
 - the transition relation $\delta^\#$ contains only transitions labeled with symbols from trace τ , i.e., $\delta^\# = \{(\ell, s_i, \ell') \mid s_i \in \tau \wedge (\ell, s_i, \ell') \in \delta\}$, and
 - the initial location $\ell_0^\#$ stays the same, i.e., $\ell_0^\# = \ell_0$.

After constructing a path program of τ , $\mathcal{P}_\tau^\#$, we compute an abstraction of $\mathcal{P}_\tau^\#$ using abstract interpretation. If the abstraction provides a proof for the infeasibility of the path program, we have obtained suitable loop invariants for all loops that occur in the path program. In this case we construct a data automaton \mathcal{A}_{AI} from $\mathcal{P}_\tau^\#$ which is then added to the existing data automaton \mathcal{A}_D . In the case where abstract interpretation fails to prove infeasibility of the path program, we generalize the trace τ with the generalization method from the trace abstraction algorithm. Therefore, our approach is able to retain the precision of trace abstraction, but has a useful mechanism to prevent divergence due to loop unrolling. In the best case, we converge faster than the trace abstraction algorithm because we are able to find suitable loop invariants for the investigated path programs.

In the following, we describe the abstract interpretation module of the algorithm in Figure 5 in more detail. The basic functionality of our abstract interpretation module is depicted in Figure 6. The abstract interpretation module is

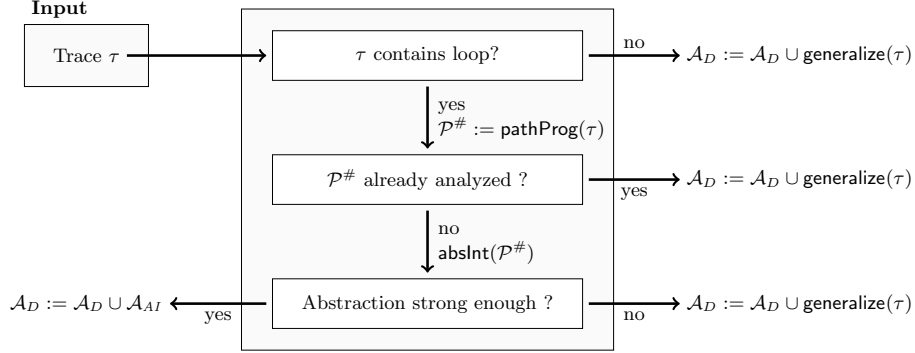


Fig. 6: Abstract interpretation module.

used when a trace τ has been identified as being infeasible. First, it is determined whether τ contains a loop. If τ does not contain a loop, trace abstraction's generalization can easily compute state assertions with the help of an SMT solver as no loop invariants are needed. If τ contains at least one loop, i.e., if there exists a statement in τ which is part of a loop in \mathcal{P} , we use τ to construct a path program, $\mathcal{P}_\tau^\#$.

Next, we check whether $\mathcal{P}_\tau^\#$ corresponds to a path program which has already been analyzed in a previous CEGAR iteration to avoid analyzing the same path program twice. This may happen if abstract interpretation was unable to find a proof for a path program in an earlier iteration and we are in the process of unrolling the loop. In this case, we have to continue with the standard trace abstraction refinement step to ensure progress. Otherwise, we use abstract interpretation to compute a fixpoint abstraction of $\mathcal{P}_\tau^\#$.

Abstract interpretation can yield two possible results: $\mathcal{P}_\tau^\#$ is proven to be safe, or the computed abstraction is too weak to prove safety. If $\mathcal{P}_\tau^\#$ is proven to be safe, i.e., the error location of $\mathcal{P}_\tau^\#$ is unreachable, the state assertions obtained through the computed abstraction are a proof for the trace's infeasibility. In this case, we construct a data automaton \mathcal{A}_{AI} from the generated state assertions which is then added to the existing data automaton \mathcal{A}_D . If the error location of $\mathcal{P}_\tau^\#$ is reachable because of a too coarse abstraction, we use trace abstraction's generalization method to generalize τ and continue with the next CEGAR iteration.

3.1 Data Automaton Construction from Path Programs

We construct a data automaton $\mathcal{A}_{AI} = (Q, \delta_{AI}, q_0, F)$ from a path program $\mathcal{P}_\tau^\# = (Loc^\#, \delta^\#, \ell_0^\#)$, annotated with state assertions obtained through the computed abstraction as follows.

- The set of locations Q of \mathcal{A}_{AI} contains a location for each unique fixpoint computed for $\mathcal{P}_\tau^\#$, i.e.,

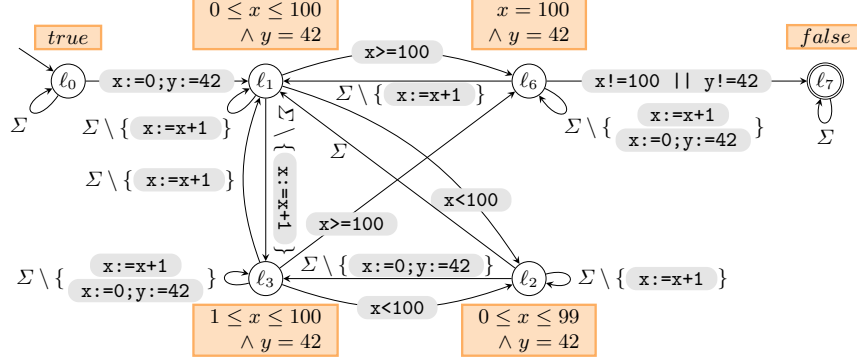


Fig. 7: Enhanced data automaton computed by an interval analysis on the path program from Figure 2 with $\Sigma = \{ \text{x:=0; y:=42}, \text{x}<100}, \text{x:=x+1}, \text{y}>0, \text{x}>=100}, \text{x!=100} \parallel \text{y!=42}, \text{y}<=0}, \text{y:=42} \}$.

- $\forall q_1, q_2 \in Q \exists \ell_1^\#, \ell_2^\# \in Loc^\# \text{ s.t. } \varphi_{\ell_1^\#} = \varphi_{q_1} \wedge \varphi_{\ell_2^\#} = \varphi_{q_2} \wedge \varphi_{q_1} = \varphi_{q_2} \iff q_1 = q_2,$
- the set of transitions of \mathcal{A}_{AI} corresponds to the set of transitions of $\mathcal{P}^\#$ with respect to the locations in Q , i.e., $\forall (q, st, q') \in \delta_{AI} \exists \ell_1^\#, \ell_2^\# \in Loc^\# \text{ s.t. } \varphi_q = \varphi_{\ell_1^\#} \wedge \varphi_{q'} = \varphi_{\ell_2^\#} \wedge (\ell_1^\#, st, \ell_2^\#) \in \delta^\#$,
 - the initial location of \mathcal{A}_{AI} and $\mathcal{P}^\#$ are the same, i.e., $q_0 = \ell_0^\#$,
 - the set of accepting states F contains the error location of $\mathcal{P}^\#$, and
 - every state $q \in Q$ is annotated with a formula φ_q which is the state assertion computed by the fixpoint engine of the path program location corresponding to q .

By construction, we retain the property of Floyd-Hoare automata, that for each transition $(q, st, q') \in \delta_{AI}$ the triple $\{\varphi_q\} \text{ st } \{\varphi_{q'}\}$ is a valid Hoare triple. Therefore, the automaton accepts at least all the traces represented by the path program.

The constructed data automaton can be further enhanced to exclude more traces by generalizing it as follows. For each triple $\{\varphi_q\} \text{ st } \{\varphi_{q'}\}$, where $q, q', st \in \mathcal{A}_{AI}$, not already represented in \mathcal{A}_{AI} it is checked whether the triple is a valid Hoare triple. We do that by computing the abstract post state of the abstract state of q and the statement st . If the resulting post state is a subset of q' , we have found a valid Hoare triple. Otherwise, the triple is not a valid Hoare triple with respect to the current abstraction. For all of those valid Hoare triples, a new transition between the corresponding states, labeled with the corresponding statements is added to \mathcal{A}_{AI} . Figure 7 shows the enhanced data automaton for the example path program from Section 1.1.

4 Implementation and Evaluation

In this section, we present the implementation and evaluation of our approach.

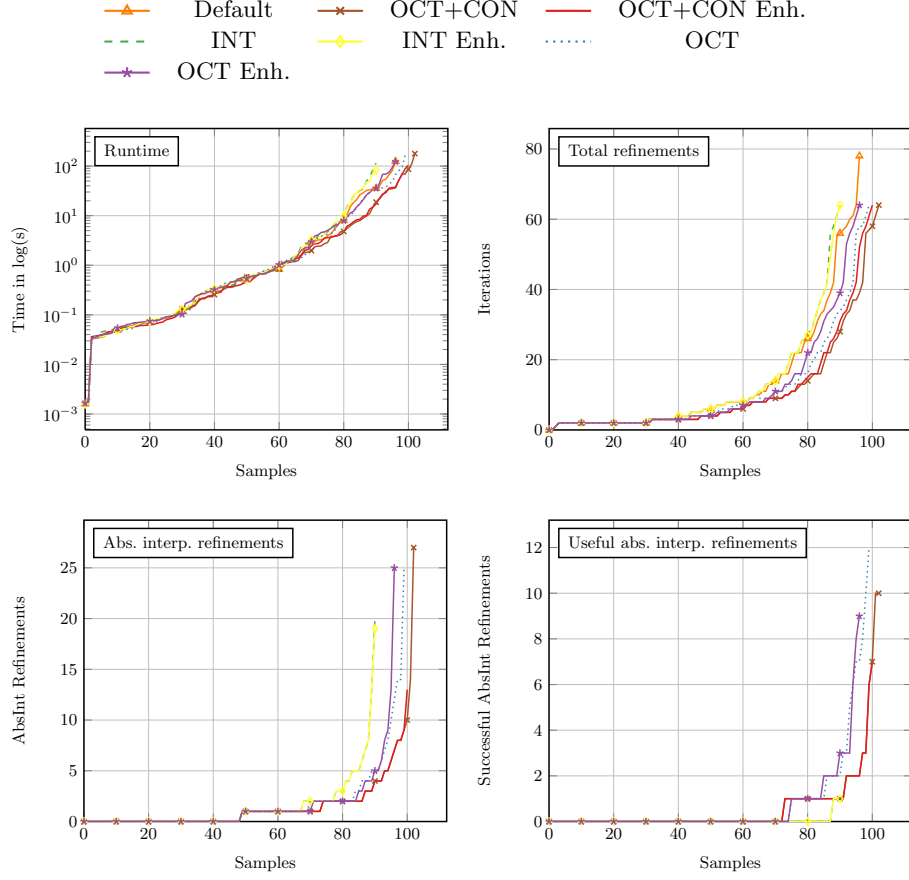


Fig.8: Statistics collected during the execution of the benchmarks. All plots show the measured data on the y-axis and range over the samples on the x-axis. The order of the samples is sorted by the measurement value for each plot. This allows us to show trends but also prevents the comparison of single samples. The upper-left chart “Runtime” compares the total runtime of the different settings. The upper-right chart “Total refinements” compares the number of iterations, the lower-left chart “Abs. interp. refinements” compares the number of iterations where abstract interpretation was applied to path programs with loops, and the lower-right chart “Useful abs. interp. refinements” shows the number of refinements where abstract interpretation computed a proof for the infeasibility of the path program.

We implemented our algorithm in `ULTIMATE AUTOMIZER`¹, a state-of-the-art software model checker which is part of the `ULTIMATE` framework². `ULTIMATE`

¹ <https://ultimate.informatik.uni-freiburg.de/automizer>

² <https://ultimate.informatik.uni-freiburg.de>

AUTOMIZER uses the trace abstraction algorithm (see Section 2) and large block encoding [5].

We also implemented an abstract interpretation engine in `ULTIMATE`. Our engine supports octagons [22] as relational abstraction, intervals and congruences [16] as non-relational abstractions, and any combination thereof. It also allows a union of different abstractions for each fixpoint, which can be parameterized.

In our experiments, we used the same settings for the `ULTIMATE AUTOMIZER` part of our approach [18] that were used when `ULTIMATE AUTOMIZER` participated in the software verification competition SV-COMP 2016 [4]. In particular, it uses `Z3` [13] to decide feasibility of a sample trace in all our experiments.

For our evaluation we applied our version of `ULTIMATE AUTOMIZER` to C programs taken from the SV-COMP 2016 [4] repository³. We used the three sample sets “Loops”, “Simple” and “ControlFlow”, which consist of a total of 237 benchmarks. We removed all examples from the directory `ssh-simplified` (category “ControlFlow”), because our abstract interpretation implementation contained a bug that prevented it from running on those examples. This left us with 214 benchmarks.

Each of the benchmarks contains one error location, which is either reachable or unreachable. For 133 benchmarks, the location is unreachable, for 81 it is reachable. The particular features of the programs of each category are as follows. “Loops” contains programs that contain multiple functions which each contain multiple possibly nested loops that manipulate the variables of the program. Note that programs in this category do not contain recursive function calls. Category “Simple” contains programs obtained by simplifying real-world examples. They contain multiple functions that are called from one single loop in the main function, and use multiple different data structures such as enumerations, structs, and unions. The “ControlFlow” category consists of programs in which multiple control variables are set within loops. The values of the control variables determines when a program enters an erroneous state. We compare the following seven different settings with each other:

- `ULTIMATE AUTOMIZER` without any modifications (Default),
- our algorithm using an interval abstraction without data automaton enhancement (INT), or with construction of the enhanced data automaton (INT Enh.),
- our algorithm using an octagon abstraction without data automaton enhancement (OCT) or with construction of the enhanced data automaton (OCT Enh.), and
- our algorithm using a combination of congruence and octagon abstraction without data automaton enhancement (OCT+CON) or with construction of the enhanced data automaton (OCT+CON Enh.)

All benchmarks were run on an Intel Core i5-3550 with 3.30GHz using a timeout of 90 seconds and a memory limit of 4GB for the tool itself and 2GB for the SMT solver.

³ <https://github.com/sosy-lab/sv-benchmarks/releases/tag/svcomp16>

	Success	Timeout	Error
Default	97 (3)	110	7
OCT Enh.	97	106	11
OCT	100	106	8
OCT+CON Enh.	101	100	13
OCT+CON	103	98	13
INT	91	104	19
INT Enh.	91	104	19
Portfolio	108	89	6

Table 1: The evaluation results. The complete benchmark set contained 214 samples. Each cell in the column “Success” contains the number of samples this particular setting could solve. The number in parenthesis shows how many samples were solved *exclusively* by this setting. The column “Timeout” contains the number of times each setting run into the 90s timeout. The column “Error” contains the number of times each setting could not solve a benchmark due to a crash of the tool. The row “Portfolio” shows how many benchmarks could be solved by any of the settings, or on how many benchmarks all settings failed or errored, respectively.

Table 1 shows the results of the evaluation. Out of the 214 input programs, the default trace abstraction implementation was able to solve 97 programs. The best abstract interpretation based CEGAR approach, OCT+CON, was able to solve 103 programs. The default configuration of ULTIMATE AUTOMIZER (Default) could solve three examples exclusively. These examples lead to timeouts in all the other settings.

Figure 8 shows various statistics of all approaches compared to each other. The top left hand chart shows the runtime in $\log(s)$ for all individual benchmark programs, ordered by time. It shows that the OCT+CON setting was not only able to prove the most programs, but also took the least time. The fact that OCT+CON could solve the most problems was not unexpected: OCT+CON computes relational constraints of the form $\pm x \pm y \leq c$, where x and y are variables and c is a constant and combines them with non-relational constraints of the form $x \bmod c = 0$. Both of these constraints are notoriously difficult to obtain for SMT solvers. Interestingly, the combination of octagons and congruence is even faster than using octagons alone (OCT). Although it is only a slight advantage, it shows that the additional information is useful in some cases.

The runtime chart also shows that the interval abstraction performed the worst of all of our settings. The reason for that is the missing precision of the interval domain compared to the octagon domain. Therefore, more iterations to prove a program are needed and the timeout occurs faster.

On the top right hand side in Figure 8 the number of refinements in the CEGAR loop is shown. This number indicates how often a new data automaton

was constructed with either the default trace abstraction algorithm or with the approach presented in this paper. Note that trace abstraction always needed to do more CEGAR iterations than the OCT, OCT Enh., OCT+CON, and OCT+CON Enh. settings. Only the interval abstraction based settings were trailing the default trace abstraction approach. This fact shows that the choice of an relational abstraction (with combination of the congruence abstraction) improves the convergence and the precision of the overall approach. Also note that about 30 programs could be proven in the first iteration of the CEGAR loop.

The lower left hand chart of Figure 8 shows the number of iterations in which a path program was constructed and analyzed with abstract interpretation. Note that in nearly half the cases, such a construction was not necessary, because the benchmarks could be solved analyzing only single traces.

The lower right hand chart shows the number of iterations in which abstract interpretation could prove the infeasibility of the path program. Compared to the total number of abstract interpretation refinements, in roughly half the benchmarks this was the case. Interestingly, there is no sample for which the interval abstraction was useful more than once, again outlining that the default variant of `ULTIMATE AUTOMIZER` can infer these invariants by itself.

The results in Table 1 and Figure 8 also show that enhanced data automata do not perform better. It seems that the checks required for adding additional edges take too much time compared to their usefulness.

5 Related Work

In their work on Craig Interpretation [1, 2], Albarghouthi et al. use a CEGAR-based approach with abstract interpretation to refine infeasible program traces. In contrast to our work, they use abstract interpretation to compute an initial abstraction of the whole program. Then, a trace to an error location is picked from the abstraction, instead of the original program, and analyzed using a bounded model checker. If the trace is infeasible, this results in a set of state assertions, which may be too precise, i.e., non-inductive, to be used to refine the initial abstraction. Abstract interpretation is used again, this time to weaken the found state assertions in an attempt to achieve inductivity before refinement of the last abstraction is done and the next iteration begins. Because the analysis is done on an abstraction dependent on the fixpoint computed by abstract interpretation, many iterations are needed in the worst case to identify infeasible program traces. The fact that we are using abstract interpretation to compute fixpoints of path programs which are a subset of the original program, instead of an abstraction, allows us to circumvent the problem that an abstraction of the whole program might be too weak to prove the program to be correct. Additionally, we often eliminate the need to use expensive model checking techniques to refine the abstraction iteratively. Therefore, our generalization with abstract interpretation is more localized and more precise than an abstraction obtained by analyzing the whole program.

Beyer et al. use path programs in a CEGAR approach to compute invariants of locations in a control flow graph of a program [8]. The refinement of the abstraction is done by using a constrained-based invariant synthesis algorithm which computes an invariant map, mapping predicates forming invariants to locations of the path program. Those invariants are excluding already visited parts from the original program. This is done until a counterexample for the program’s correctness has been found or the program has been proven to be correct. In contrast to our work, their approach uses an interpolant generator to generate the invariant mapping, whereas we use both, an interpolant generator and a fixpoint computation engine to obtain suitable state assertions. In addition, their approach is only able to synthesize loop invariants by using invariant templates which are parametric assertions over program variables, present in each location of the program. Although they propose to use other approaches to generate invariants, including abstract interpretation, they do not present a combination of those methods.

6 Conclusion

In this paper, we presented a CEGAR approach that benefits from the precision of trace abstraction and the scalability of abstract interpretation. We use an automata theoretical approach to pick traces from a program automaton which are checked for infeasibility. If the trace is infeasible, we construct a path program and compute an abstraction of the path program by using abstract interpretation. With the help of this abstraction, we are guaranteed to obtain state assertions, in particular loop invariants, which help us to exclude a generalization of the found infeasible trace from the program. Because abstract interpretation may yield an abstraction which is not precise enough to synthesize usable loop invariants, we use the default precise trace abstraction approach as a fallback.

Our experiments show that by using abstract interpretation to generate loop invariants of path programs, we are not only able to prove a larger set of benchmark programs, but also need less CEGAR iterations to do so, leading to a more efficient approach to proving correctness of programs.

References

- [1] A. Albarghouthi, A. Gurfinkel, and M. Chechik. Craig Interpretation. In *SAS 2012*, pages 300–316. Springer, 2012.
- [2] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. Ufo: A Framework for Abstraction- and Interpolation-Based Software Verification. In *CAV 2012*, pages 672–678, 2012.
- [3] T. Ball and S. K. Rajamani. The SLAM Toolkit. In *CAV 2001*, pages 260–264, 2001.
- [4] D. Beyer. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016). In *TACAS 2016*, pages 887–904, 2016.

- [5] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software Model Checking via Large-Block Encoding. In *FMCAD 2009*, pages 25–32. IEEE, 2009.
- [6] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker BLAST. *STTT 2007*, 9(5-6):505–525, 2007.
- [7] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *VMCAI 2007*, pages 378–394, 2007.
- [8] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path Invariants. In *PLDI 2007*, pages 300–309, 2007.
- [9] D. Beyer and M. E. Keremoglu. CPACHECKER: A Tool for Configurable Software Verification. In *CAV 2011*, pages 184–190, 2011.
- [10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided Abstraction Refinement. In *CAV 2000*, pages 154–169, 2000.
- [11] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL 1977*, pages 238–252, 1977.
- [12] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL 1978*, pages 84–96, 1978.
- [13] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS 2008*, pages 337–340, 2008.
- [14] D. Dietsch. *Automated Verification of System Requirements and Software Specifications*. PhD thesis, University of Freiburg, 2016.
- [15] R. W. Floyd. Assigning Meanings to Programs. *Mathematical Aspects of Computer Science*, 19(19-32):1, 1967.
- [16] P. Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT 1991*, pages 169–192, 1991.
- [17] M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski. Ultimate Automizer with SMTInterpol - (Competition Contribution). In *TACAS 2013*, pages 641–643, 2013.
- [18] M. Heizmann, D. Dietsch, M. Greitschus, J. Leike, B. Musa, C. Schätzle, and A. Podelski. Ultimate Automizer with Two-track Proofs - (Competition Contribution). In *TACAS 2016*, pages 950–953, 2016.
- [19] M. Heizmann, J. Hoenicke, and A. Podelski. Refinement of Trace Abstraction. In *SAS 2009*, pages 69–85, 2009.
- [20] M. Heizmann, J. Hoenicke, and A. Podelski. Software Model Checking for People Who Love Automata. In *CAV 2013*, pages 36–52, 2013.
- [21] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.
- [22] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
- [23] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL 1999*, pages 105–118, 1999.
- [24] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI 2006*, pages 25–41, 2005.